UTM UNIVERSITI TEKNOLOGI MALAYSIA
INTERNATIONAL JOURNAL OF INNOVATIVE COMPUTING
ISSN 2180-4370
Journal Homepage : https://ijic.utm.my/

Enhancing the Developer Experience (DX) in Docker Supported Projects

Masitah Ghazali*
Malaysia – Japan International Institute of Technology (MJIIT), Universiti Teknologi Malaysia, Jalan Sultan Yahya Petra,54100 Kuala Lumpur, Malaysia
Email: masitah@utm.my

Alfian Naufal Ravi Hidayat
Faculty of Computing
Universiti Teknologi Malaysia,
81310 UTM Johor Bahru, johor, Malaysia
Email: naufalravi@graduate.utm.my

Submitted: 13/11/2022. Revised edition: 15/2/2023. Accepted: 15/2/2023. Published online: 30/5/2023
DOI: https://doi.org10.11113/ijic.v13n1.393

***Abstract*—Docker is undeniably powerful and revolutionary in how containerized system development is developed today, but it is apparent that the learning curve for it should be addressed, as it typically is complex at times, especially for beginners. One of the fundamental tasks in a Docker workflow is Dockerfile configurations, which at times require ample time to study and observe for attaining the best practices, even the appropriate result. This issue undeniably affects the developer experience. Developer Experience (DX), being a derived field from User Experience (UX) that has been getting traction for the past few years concerns developers' innate ability to perceive tasks as enjoyable, painful, or perhaps some other sets of emotions. The goal of DX is to evaluate all those factors in order to improve the software development experience, which consequently affects how the project is delivered. In resonance with that, this work aims to enhance the DX by way of proposing and incorporating supporting interaction tools, both based on CLI and GUI as the interface type, with two different permutations: CLI and GUI. The DX of both has to be evaluated by the experts, who are of experienced developers, regardless of whether they have the knowledge of Docker or not. The method to test and evaluate two different solutions is conducted qualitatively, with each respondent had a different order of evaluating the two solutions. The qualitative data is thematically analyzed, resulting in GUI being the best option among the two. The contribution of this research is the design guidelines for GUI and CLI-based tools development that enhance the Developer Experience (DX) in the scaffolding of Dockerfile and docker-compose.yml for projects that use Docker.**

***Keyword*—Developer Experience, Docker, Command-line Interface, Graphical User Interface, Qualitative Analysis**

I. INTRODUCTION

Containerization is a state of the art of development, and Docker is one of the most popular solutions. It solves dependency management issues, conflicts between local environments, and resources being utilized, which in consequence, makes deployment and collaborative development less problematic. The way it is configured is in two ways, one is by defining the Dockerfile, which is typically used to assemble a single image, and the second is the docker-compose.yml, where both single and multi-interconnected-containers are meant to be configured. The fact that an ample amount of time is required to learn to start configuring containers from scratch, especially with docker-compose is undeniable. There are also concerns regarding implementing one approach over the other, which could result in unforeseen implications, such as security, for instance. This often raises concern about developer experience (DX), as consequence, frustration increases among developers and students, which is not a sign of good developer experience [1].

We aim to understand the developers experience as they use the Docker application. The primary objective of this study is to evaluate the effectiveness of the interaction techniques and identify the one that works best for the developer experience (DX) on the current Docker files configuration. Section II discusses the existing research on DX, Docker, CLI, and GUI. Section III Discusses the method that this study shall follow, and Section IV describes the Proposed Guidelines and User Design Setup. Section V is Results and Analysis, and finally, Section VI is Conclusion.

II. Related Work

A. Developer Experience (DX)

DX is a relatively new field, as the oldest and the pioneer of research about it, Fagerholm [2] describes the need to create a new terminology specific for developers, which subsequently creates a new research field concerned with enhancing developers' software development process efficiency concerning their inherent emotions. The direction of Fagerholm [2] and the idea of creating a terminology can clearly be seen, as it emphasizes the notion of making the developers and their inherent emotions being the main focus and study subject, as the literature also mentions that there are numerous productivity factors, which most of them being non-technical. The literature describes alternative ways to perceive DX, which could be done by breaking down each word, but ultimately it all comes down to user experience and psychology, two domains that are interrelated. UX, being the influence of DX itself is quite new, being under the domain of HCI, having the characteristics of dynamic, context-dependent, and subjective [3].

B. DX and Psychology

Intellect, or cognition, is a part of one of the oldest models in cognitive psychology, the tripartite classification of mental activities, cognition, affection, and conation [4]. Although there are disagreements and whatnot prior to the literature being published, Hilgard put a lot of emphasis on the prominence of such a model for the assessment of contemporary emphases in psychology. As for the case of DX, the scheme of tripartite of mind could support the argument that emotions and cognition are then turned into intentional actions being done during the software development process [2].

C. Containers and Docker

Containerization is a way to create virtualization at the OS level, which means the containers are using the same OS as the machine, in a way, operating on top of it [5]. This way, it offers isolation of the filesystem, while having its own resources shared by the host OS, hence the term containers.

While Docker and the term containerization have been popular lately, the first pervasively-used container is LXC, or Linux Containers [6]. LXC allocates resources as necessary using Cgroups. Cgroups or Control Groups are resource controllers for processes that reside in an operating system, initiated by Google in 2007, and proceeded to be implemented in the main Linux kernel in 2008 [7]. Each container within LXC has its own kernel, which is shared with the host OS, making the processes accessible to it [6].

Docker offers extensive features to LXC, having features that LXC has while having more kernel and application-based features in order to make data management possible on top of the host OS [6]. Docker containers are the result of the blueprint created in the shape of Docker Images. Images could be an OS, like Linux, a Database Management System (DBMS), or any complex applications or platforms that are already bootstrapped in terms of configuration and all the complex bits, which makes them ready to be utilized instantly [5].

D. Interface and Interaction Types

As history goes, CLI was conceived first, as a way to communicate with computers by imperative commands in natural language. As the system grew, more commands were introduced, hence more effort was needed to memorize, hence GUI is there as an alternative to serve as an abstraction by visualizing steps that the command would reproduce by selecting sequences of objects that describe such [8]. GUI exists as an alternative to the conventional way to interact with computers at that time, through a blank screen with a prompt, where the user could enter commands that are already pre-defined [9]. At that time, the term GUI itself had no exact meaning, as studies were limited. There were multiple terms being brought out, namely by Harding [10] and Bonsiepe [11], where both are associating GUI with the looks or visualization of the computer system and its interactions.

III. Method

There are four phases in the research workflow, the first being requirements gathering, followed by requirements specification, then prototype design, and finally evaluation. The whole workflow is not meant to be linear, as the last two are done iteratively. Fig. 1 shows the research workflow, with the leftmost being the first phase, incrementing to the right, up to four, that denotes successive processes.

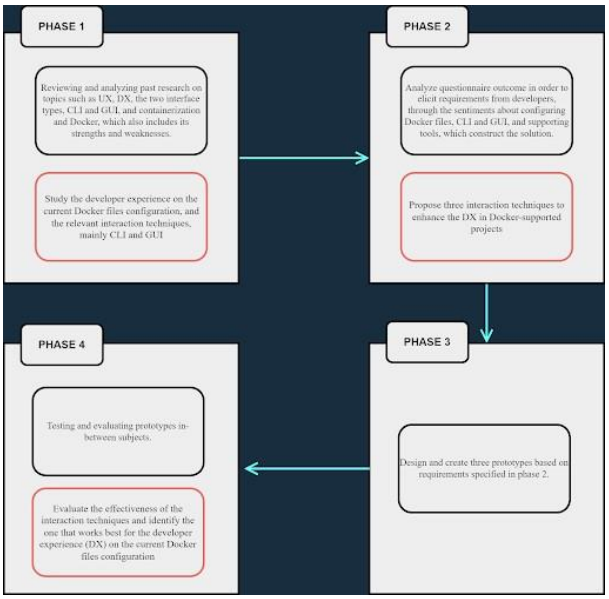

Fig. 1. Research workflow

A. Phase 1: Acquiring Information

The first phase is Acquiring Information, where past research or any relevant literature is reviewed in order to collect data and foundational information on topics namely

UX, DX, interface types, and containerization, which leads up to Docker. Apart from that, a questionnaire of 19 questions is formulated in order to elicit sentiments, enjoyments, and pain points that developers and students alike would get when working with Docker, especially its ultimate task, images configuration, while also touching upon the same points with interface types, and overall sentiment on supporting tools. This aligns with the first objective, which is to study the developer experience with the current Docker files configuration, and the relevant interaction techniques.

B. Phase 2: Design Proposal

The following phase is the Design Proposal, where data analysis is done based on the data collected from the questionnaire in the previous phase. It is done through qualitative analysis, of a grouping of answers that could lay the basis for prototype designing, by proposing the most appropriate solution based on actual users' feedback. This helps satisfy the second objective, which is to propose two interaction techniques to enhance the DX in Docker-supported projects.

C. Phase 3: Prototype Design

Next is Prototype Designing, where the activity is as it suggests, design prototypes, which are for two different interface types, CLI and GUI based on the data gathered in the first phase and analyzed in the second. Prototype designing is done iteratively, as evaluation from developers is required as Kuusinen [12] suggests, to achieve the most fitting DX for a particular software or in this context, the tools that are being proposed in order to enhance the DX of Docker-supported projects. To achieve this, co-designing sessions are conducted.

D. Phase 4: Identifying the Most Suitable Technique

The last phase is to identify the most suitable technique. An experiment will be conducted by recruiting the same participants in the second phase. In other words, the test is done in-between subjects. The evaluation itself is going to be done right after an initial working version of the system is ready. This goes along with Kuusinen's suggestion [12], as in the 2016 study, the process was well-liked. A concern regarding bias on the part of evaluators, by preferring the last testing object just because of a particular order, arises, but this could be minimized by dividing the participants into three separate groups, with different orders. From there on, the third objective could be achieved, which is to evaluate the effectiveness of the interaction techniques and identify the one that works best for the developer experience (DX) on the current Docker files configuration.

IV. Analysis and Proposed Improved Designs

A. Insights

Respondents were invited to participate in a survey, where some insights were learned about their experiences while using the Docker application, the challenges they often face, and their views and feedbacks, both positively and negatively on the styles of interfaces, particularly the graphical user interface (GUI) and the command line interface (CLI). A total of 19 respondents took part in the survey and their responses are as shown in Table I, II and III.

TABLE I. Enjoyment while Using Docker

Docker's Factual Advantages	Existing External Supporting Elements	Docker Inherent Tasks
• The ability to create a reproducible development environment • Familiarity with Linus, hence Docker environment configuration made easy • Human readable and works out of the box • Easy and lightweight to setup • Seamless multi-platform integration with the same configuration	• Docker documentation • Resources and documentation provided by Docker community • Community support, tooling, and bootstrapped complex configuration	• Define multiple images, and networking for connecting all those images • Configuring existing images and exposing ports • Configuring a new container • Creating and adding commands to Dockerfile • Configuring container's name

TABLE II. Challenges while Using Docker

Learning Curve	Docker Inherent Tasks
• First time adopting Docker • There should be a one-click setup analogous to NodeJS that offers an easier installation method	• The need for a debugger and autocompletion for Dockerfile and docker-compose • Connecting backend and DB containers • Configuring default parameter for mountable drive for example, since knowing the arguments are necessary • When working with multiple images • Configuring volumes, custom Docker images environmental Docker files such as docker-compose.local • Image size and permission-related issue • No support for VM • Unrestricted access of processes and files • Configuration format and all available options that could reduce security holes

TABLE III. FEEDBACK ON CURRENT INTERFACE TYPES

Positive	
Command Line Interface (CLI)	**Graphical User Interface (GUI)**
• Lightweight • Efficient and able to be automated • Reduced context-switching • Dynamic • Straightforward • Commands and arguments are able to be combined, then executed as a script	• Tends to be user-friendly • More intuitive and interactive • Errors recognition is easier • Feedback is visual, hence recognized faster compared to text • Shortcuts option
Negative	
Command Line Interface (CLI)	**Graphical User Interface (GUI)**
• Hard to navigate while dealing with dynamic data • Limited visualization • Remembering all options and arguments are required • Steeper learning curve • Obscure error codes that require searching online • Knowing available configurations is required • Verbose commands • Inconsistent commands	• Lack of flexibility • Automation is difficult compared to CLI • GUI is cluttered or easy to be disorganized most of the time • Process, memory and storage demanding • Too many GUI features could also translate to distraction and requires learning • Obscure representation of objects in GUI

Based on the data gathered, most of the respondents are inclined towards the usage of CLI, though it is evident that GUI also shares quite a fair number of likings. In addition, the documentation, being a rudimentary part of a system, as the questionnaire result suggests for a web-hosted is preferred as the medium of accessing it. Therefore, in enhancing the DX, the improved design for both CLI and GUI interface types are based on the findings from the literature review, questionnaire answers, and co-designing sessions.

B. Proposed Design

1) Command Line Interface

The solution is designed for flexibility, as CLI-based tools require commands to perform certain tasks, which are operable through the terminal. Through it, the user would be able to enter commands that could create default configurations for Dockerfile or docker-compose. For example, entering "MySQL", "PHP-FPM", and "Redis" would generate a docker-compose file that has configurations that the documentation for each image would recommend. Some typical ones are port numbers, and volumes, which sometimes could hinder one's development speed, especially for beginners, as learning is required. The goal is to abstract what to configure and modify. Documentation is definitely required, therefore adding information on what a particular command does, or perhaps what arguments and flags should be passed could be accessed through its manual or help page. The proposed and improved working CLI is made in Golang, which the binary then will be distributable.

2) Command Line Interface

The GUI-based solution serves as a bootstrapped Dockerfile and docker-compose configuration, similar to what CLI can do, but is done through a web application that offers click and select options, which results in a downloadable file. From there on, modification is done without help, as it only serves as a one-time abstraction.

Figma is used to create the working GUI interactive prototype. Previously, the mockups or the rough sketches were built based on the survey findings, as well from the design decisions that were made solely based on the use cases. Fig. 2 shows the initial design of the Dockerfile Configuration.

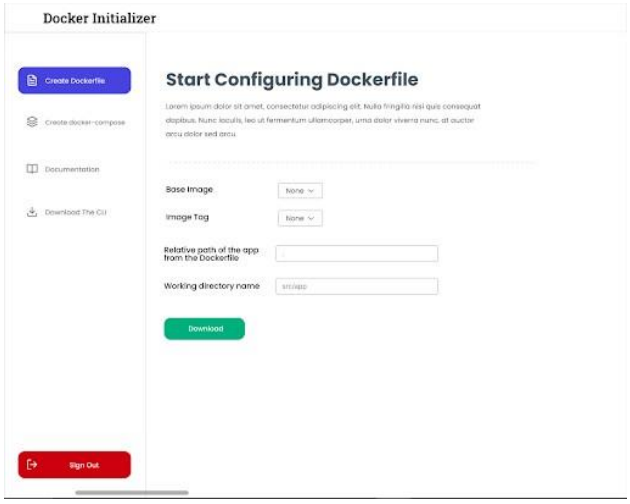

Fig. 2. Initial GUI design of Dockerfile Configuration

It is imperative to obtain initial feedback from the targeted potential users; developers and Dockers users alike, on the suggested improved GUI design. For that, co-designing is chosen to be a way to transform user's feedback into a product that shall be beneficial to the targeted users. In fact, co-designing promotes better creative process during the design, while also making service definition clearer. Not only that, it also provides satisfaction from the user's side, while also fitting what the user really requires [13]. In this process, two individuals with different roles and experiences (see Table IV) volunteered to provide feedback and insights on the initial design, where they helped in improving the design together.

TABLE IV. ROLES AND EXPERIENCE OF THE CO-DESIGNERS

	P1	**P2**
Role	Site Reliability Engineer	Student (exposed and experienced as DevOps engineer)
Experience in using Docker application	Using it in production, intermediate to advanced level	Familiar

V. COMPARISON STUDY

The proposed and improved interface types; the CLI and the GUI, as previously described are then compared to identify the most suitable type that could enhance the developer experience (DX) in the Docker related application. A comparison study is designed in such a way that the participants can experience both types to test which is most comfortable to them.

Five participants involved in this study. They were first briefed about the project and its aim, and were asked to complete a consent form. Once they are comfortable, XX tasks were handed out to them to accomplish at their own. All participants used their own personal computer in either a UNIX-based environment, or Windows subsystem for Linux.

This study is a within-subject study where all five of them tested both conditions, i.e. both interface types. In order to reduce the order effect, counter balancing was performed, where two of them started with GUI, and the other with CLI, and vice versa. The three tasks, described below, were the same for both conditions.

- Task 1: Complete the sign up and sign in
- Task 2: Configure a Dockerfile of image node, with the entrypoint of app.js, env variable of NODE_ENV=production
- Task 3: Configure a docker-compose of stack MERN, set the project name to "my-beautiful-project", set node:latest as the tag, and index.js as the entrypoint. The environment variable for node should be NODE_ENV=development, and for mongo:
  username: root
  password: extrasecret

While performing the tasks, the participants were encouraged to think aloud in order to know their thought process. Their behaviour was also being observed throughout the session. At the end of the session, the participants were interviewed guided by the following four questions.

- What was difficult?
- What motivated you to use both the CLI and GUI?
- What could be your biggest concern with the CLI-based tool? And what about the GUI-based?
- What do you think about the learning curve of the GUI-based, and the CLI-based?
-

VI. RESULTS

The findings from the tasks accomplishment, observations and the interview were compiled and analysed. All participants managed to do all tasks assigned in both conditions.

With regards to the questions posed during the interview, interesting enough, all participants find the CLI type to be the most challenging. Participant P3 even quoted, "*I needed just a little bit of docs of what it was expecting. It was great, and I tried -h flags …. and after a while, it was much easier to create"*. Noting that the documentation provided for the root command was minimal, making all participants to explore on their own. The lack of documentation in that aspect that made all participants starting out using the tool to feel difficult while using seems to be true, since the rest, which was well-documented was a smooth sailing experience, especially for task number 3. Apart from that, an interesting remark that P1 said, "*There are too many flags, but yet they are compulsory to use. Are they not supposed to be commands if they are must to have? Regardless, there are so much to write, and it took too long"*. P3 also had the same idea when it comes to the command required to run is too long.

There were no complains about the GUI, as all participants unanimously liked the straightforwardness and how easy it was for first time users. Though P5 let out a concern, *"I feel like the GUI is straightforward, but there are some cases where I had to type in a long text just to pass an argument. Would not that be nice if there are some assistive components that could make it faster and easier."*

The second question was mainly answered in a way, appreciating the tools that could be used to make Dockerfile and docker-compose.yml configuration simpler, but yet, they appreciated the GUI more, as it did make everyone felt easier to get used to, faster to use, and efficient in terms of task completion. Although the GUI has no documentation at all, but the UI components and the placeholders could show what the system is expecting, which everyone agreed. P3 quoted, *"The placeholder made it more clear to what the GUI-based tool should be expecting"*. P4 said, "*GUI is straightforward and predictable, so the user knows what the possible usage are. Also, transitioning from CLI to GUI was not difficult."*.

When asked about the biggest concern that they had on both interface types, CLI had the most critics, where the participants saw the issue in different ways than one: portability, efficiency, lack of documentation, and design. Design was only mentioned by P1, where the concern was mainly on the design decisions, using flags rather than sub-commands. But portability, efficiency, and the lack of documentation was the common theme. Portability was mentioned by P1, P2, and P5, where the issue was not having CLI installation to be seamless, requiring the right OS and a set of additional steps such as changing the file permission to be executable. P1 and P5 mentioned about the necessity of onboarding, but P1 also said, *"This seems to be demotivating as there are extra steps required to run the CLI. Most of the time, using such tools should be a one-off, hence making it a bit inefficient considering the effort to install."*

Lastly, on the last question, all participants are also concerned about the steep learning curve of the CLI. P4 mentioned, *"I prefer CLI personally, but the lack of documentation thereof makes me second-guess commands and flags."* As the opposite, GUI won the favor of everyone in terms of the learning curve, where P5 said, *"... GUI only requires interaction with the UI components, as for example opening the dropdowns to see the options, while the CLI requires consultation to the documentation."*

Based on the feedback attained from the interviews, we are able to further analyse them into themes. This is shown in Table V.

TABLE V. IDENTIFIED THEMES

Main Theme	Sub-themes
Inherent feelings when facing difficulties using either interface type	• Positive inherent feelings when not facing any difficulties when using GUI • Negative inherent feelings when using CLI
Inherent feelings regarding the motivation of using both types	• Positive inherent feelings when using GUI • Negative inherent feelings when using CLI
Inherent feelings regarding sentiment on both types	• Positive inherent feelings regarding the sentiment of GUI • Negative inherent feelings on the sentiment of CLI
Inherent feelings regarding the learning curve of both	• Positive inherent feelings regarding the learning curve of GUI • Negative inherent feelings regarding the learning curve of GUI

It could be inferred that the users prefer GUI over CLI, even the results of the feedback of the former eclipses the latter. Though to note, the negative feelings are not utterly unanimous for CLI, but the number still lopsided. It can be inferred that GUI is more preferred to be chosen as the interface type for bootstrapping Dockerfile and docker-compose configuration, but there are elements that CLI could improve, namely in terms of the design, portability, and documentation.

VII. CONCLUSION

It could be inferred that the users prefer GUI over CLI, even the results of the feedback of the former eclipses the latter. Though to note, the negative feelings are not utterly unanimous for CLI, but the number still lopsided. It can be inferred that GUI is more preferred to be chosen as the interface type for bootstrapping Dockerfile and docker-compose configuration, but there are elements that CLI could improve, namely in terms of the design, portability, and documentation. Future works could be in the form replicating the study to reach more developers while also improving the prototypes, in terms of the interactiveness, meaning that using a proper language and practices for writing a web application instead of a interactive design, and also proper documentation and user friendliness, since when it comes to CLI, the issues was mainly regarding the documentation, portability, and the design. Perhaps studies in the future should add better documentation, offers scripting or any other way to make installing and building in multiple OSs easier, and using the best practices for writing a CLI, in terms of the design.

ACKNOWLEDGMENT

We would like to thank the volunteers in the study, for their involvement as participants, co-designers and testers.

REFERENCES

[1] B. Reselman. (2020). Developer experience: An essential aspect of enterprise architecture. In Enable Architect. Retrieved July 27, 2022, from https://www.redhat.com/architect/developer-experience.

[2] F. Fagerholm, and J. Munch. (2012). Developer experience: Concept and definition. *International Conference on Software and System Process (ICSSP).* https://doi.org/10.1109/icssp.2012.6225984.

[3] E. L.-C. Law, V. Roto, M. Hassenzahl, A. P. O. S. Vermeeren, and J. Kort. (2009). Understanding, scoping and defining user experience. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/1518701.1518813.

[4] E. R. Hilgard. (1980). The trilogy of Mind: Cognition, affection, and Conation. *Journal of the History of the Behavioral Sciences*, *16*(2), 107-117. https://doi.org/10.1002/1520-6696(198004)16:2<107:aid-jhbs2300160202>3.0.co;2-y.

[5] D. Bernstein. (2014). Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, *1*(3), 81-84. https://doi.org/10.1109/mcc.2014.51.

[6] A. Kovacs. (2017). Comparison of different Linux containers. *40th International Conference on Telecommunications and Signal Processing (TSP).* https://doi.org/10.1109/tsp.2017.8075934.

[7] S. S. Kumaran. (2017). Introduction to Linux Containers. In: Practical LXC and LXD. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-3024-4_1.

[8] D. Norman. (2007). The next UI breakthrough: command lines. *Interactions*, *14*(3), 44-45

[9] B. J. Jansen. (1998). The graphical user interface. *ACM SIGCHI Bulletin*, *30*(2), 22–26. https://doi.org/10.1145/279044.279051.

[10] B. A. Harding. (1989). Windows & Icons & Mice, Oh My! The Changing Face of Computing. *Frontiers in Education Conference 1989:19 th Annual*. 337-342.

[11] G. Bonsiepe. (1993). Interpretations of Human User Interface. *Visible Language*, *24*(3), 262-285.

[12] K. Kuusinen. (2016). Are software developers just users of development tools? assessing developer experience of a graphical user interface designer. *Lecture Notes in Computer Science*, 2016, 215-233. https://doi.org/10.1007/978-3-319-44902-9_14.

[13] M. Steen, M. Manschot, and N. De Koning. (2011). Benefits of co-design in service design projects. *International Journal of Design*, *5*(2), 53-60.